SmartMiner: A Depth First Algorithm Guided by Tail Information for Mining Maximal Frequent Itemsets

Qinghua Zou
Computer Science Department
University of California-Los Angeles
zou@cs.ucla.edu

Wesley W. Chu
Computer Science Department
University of California-Los Angeles
wwc@cs.ucla.edu

Baojing Lu
Computer Science Department
North Dakota State University
baojing.lu@ndsu.nodak.edu

ABSTRACT

Maximal frequent itemsets (MFI) are crucial to many tasks in data mining. Since the MaxMiner algorithm first introduced enumeration trees for mining MFI in 1998, there have been several methods proposed to use depth first search to improve performance. To further improve the performance of mining MFI, we proposed a technique to gather and pass tail (of a node) information to determine the next node to explore during the mining process. Our algorithm uses an augmented dynamic reordering heuristic with considering of the tail information. Compared with Mafia and GenMax, SmartMiner generates a much smaller search tree, requires a smaller number of support counting, and does not require superset checking. Using the datasets Mushroom and Connect, our experimental study reveals that SmartMiner generates the same MFI as Mafia and GenMax, but yields an order of magnitude improvement in speed.

Keywords

Data mining, frequent patterns, maximal frequent pattern, tail information, search space pruning.

1. INTRODUCTION

Mining frequent itemsets in large datasets is an important problem in the data mining field since it enables essential data mining tasks such as discovering association rules, data correlations, sequential patterns, etc. The problem of finding frequent itemsets was originally proposed by Agrawal [1] in his association rule model and the support confidence framework. It can be formally stated as following:

Let I be a set of items and D be a set of transactions, where a transaction is an itemset. The support of an itemset is the number of transactions containing the itemset. An itemset is frequent if its support is at least a user specified minimum support value, minSup. Let FI denote the set of all frequent itemsets. An itemset is closed if there is no superset that has the same support. The set of all frequent closed itemsets is denoted by FCI. A frequent itemset is called maximal if it is not a subset of any other frequent itemset. We denote MFI as the set of all maximal frequent itemsets. Any maximal frequent itemset X is a frequent closed itemset since no nontrivial superset of X is frequent. Thus we have $MFI \subseteq FCI \subseteq FI$.

There are three different approaches for generating FI. First, candidate set generate-and-test approach [1,11,14,8,12,7]: most previous algorithms belong to this group. The basic idea is to generate and then test the candidate set. This process is repeated in a bottom up fashion until no candidate set can be formed. Second, sampling approach [7]: it selects samples of a dataset to form the candidate set. The candidate set is tested in the entire dataset to identify frequent itemsets. Sampling reduces computation complexity but the result is incomplete. Third, data transformation approach [6,16,17]: it transforms a dataset for efficient mining. For example, the FP-tree [6] builds up a compressed data representation called FP-tree from a dataset and then mines frequent itemsets directly from the FP-tree. The pattern decomposition algorithm (PDA) [16,17] decomposes transactions and shrinks the dataset in each pass. Both FP-tree and PDA greatly reduce the original dataset and also do not need to generate candidate sets.

When the frequent patterns are long, mining FI is infeasible because of the exponential number of frequent itemsets. Thus, algorithms mining FCI [9,15,10] are proposed since FCI is enough to generate association rules. However, FCI could also be exponentially large as the FI. As a result, researchers now turn to find MFI. Given the set of MFI, it is easy to analyze many interesting properties of the dataset, such as the longest pattern, the overlap of the MFI, etc. All FI can be built up from MFI and can be counted for support in a single scan of the database. Moreover, we can focus on part of the MFI to do supervised data mining.

In this paper we introduce the SmartMiner that at each step passes tail information (defined in section 2) to guide the search for new MFI. SmartMiner using an augmented heuristic and tail information has many benefits: it does not require superset checking, reduces the computation for counting support, and yields a small search tree. Our experimental results reveal that SmartMiner is an order of magnitude faster than Mafia [4] and GenMax [5] in generating MFI on the same datasets.

1.1 Related works

We first introduce an enumeration tree for an itemset *I*. Assume there is a total ordering $\leq_L$ over the items *I* in the database. We say $i_j \leq_L i_k$ if item i_j occurs before item i_j in the ordering. This ordering can be used to enumerate the item subset lattice (search space). Each node composed of head and tail represents a state in the search space. The head is a candidate for FI while the tail contains candidate items to form new heads. For example, Figure 1 shows a complete enumeration tree over five items *abcde* with the ordering *a,b,c,d,e*. Each node is written as *head:tail*. It begins with root node *:abcde*. For each item a_i in the tail of a node *X:Y*, a sub node is created with Xa_i as its head and the items

after a_i in Y as its tail. For instance, the head of the node *:abcde* is empty and its tail is *abcde*; the head of *b:cde* is *b* and its tail is *cde*.

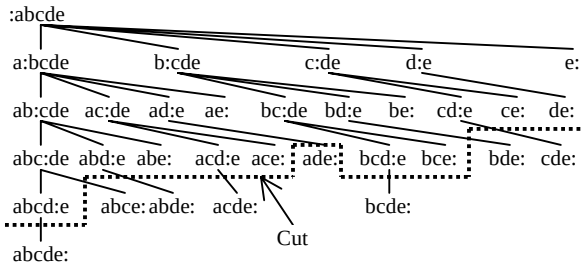

Figure 1: An enumeration tree for *abcde* for the given order of *a, b, c, d, e*

The problem of mining frequent itemsets is to find a cut through this lattice such that all itemsets above the cut are frequent, and those below the cut are infrequent (see Figure 1). A node is called a frequent node if its head is frequent. The positive border consists of the frequent nodes directly above the cut, while the negative border is the set of infrequent nodes directly below the cut. With a simple traversal without pruning, we need to count the supports of all nodes above the cut and also the negative border.

Using the enumeration tree as shown in Figure 1, we can describe recent approaches to the problem of mining MFI. MaxMiner [3] uses a breadth-first search and performs look-ahead pruning on tree branches. The look-ahead use superset pruning, i.e., if the head of a node with its tail is frequent, there is no need to further process the node since all descents of the node will be frequent. MaxMiner also first introduced the heuristic that is to reorder items in the tail of a node in the increasing order of their support. This technique is known as dynamic reordering. In general, however, superset pruning works better with a depth-first approach since many long frequent itemsets may already have been discovered. But MaxMiner uses a breadth-first approach to limit the number of passes over the database. Since large main memory size is available (in Gigabyte), depth first search is used to efficiently find long patterns.

DepthProject [2] uses depth first search on a lexicographic tree of itemsets to find MFI, and projects transactions database on the current node to speed counting the support of itemsets. DepthProject also use the look-ahead pruning and dynamic reordering. With dynamic reordering, infrequent items at the current node can be deleted from the tail so that the size of the search space can be greatly reduced.

Mafia [4] proposes parent equivalence pruning (PEP) and differentiates superset pruning into two classes FHUT and HUTMFI. For a given node *X:aY*, the idea of PEP is that if *sup(X)=sup(Xa)*, i.e. every transaction containing *X* also contains the item *a*, then the node can simply replaced by *Xa:Y*. The FHUT is to use leftmost tree to prune its sister, i.e., if the entire tree with root *Xa:Y* is frequent, then we do not need to explore the sisters of the node *Xa:Y*. The HUTMFI is to use the known MFI set to prune a node, i.e., if itemset of *XaY* is subsumed by some itemset in the MFI set, the node *Xa:Y* can be pruned. Mafia also uses dynamic reordering to reduce the search space. The results show that PEP has the biggest effect of the above pruning methods (PEP, FHUT, and HUTMFI) and dynamically reordering the tail also has dramatic savings.

Both DepthProject and Mafia mine a superset of the MFI, and require a post-pruning to eliminate non-maximal patterns [5]. Algorithm GenMax [5] integrates pruning with mining and returns the exact MFI by using two strategies. First, just like transaction database is projected on current node, the discovered MFI set can also be projected on the node and thus yields fast superset checking. Second, GenMax uses Diffset propagation to perform fast frequency computation. Experimental results show that GenMax has comparable performance with Mafia.

1.2 Limitations of Previous Algorithms

For simplicity, we use Mafia as an example to illustrate problems that existed in previous approaches. For the example in Figure 1, Mafia will generate a search tree, as shown in Figure 2, assuming that frequent itemsets have different support and the nodes are already sorted in the order of increasing support. In the figure, the

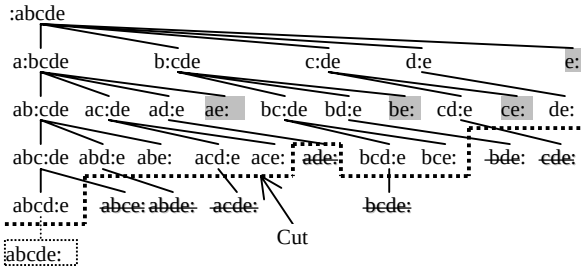

Figure 2: The search tree for Mifia with dynamic reordering and the three pruning techniques.

shaded nodes will be removed by superset pruning. The node *abcde:* in dotted box is not a part of the search tree since dynamic reordering is used. The nodes with lines crossing through are tested and found to be infrequent.

First, the size of the tree is too big and can be reduced. Although the shaded nodes can be pruned away, a more efficient strategy is not to generate those nodes in the search tree. In Figure 2, Mafia traverses 31 nodes. SmartMiner uses such a strategy and traverses only 9 nodes (see section 3.2) for the same example.

Second, there exists too much support counting for determining the frequency of tail items. Figure 3 shows the tree for counting support for Figure 2. Let X be an itemset and $T(X)$ be the set of transactions than contains X. For the root node at the top level, the transaction set is $T(\phi)$ since the head of the node is empty ϕ. For the node, the supports of *a,b,c,d,e* are counted and found to be above *minsup*. In the transaction set $T(a)$, we found *b,c,d,e* to be frequent.

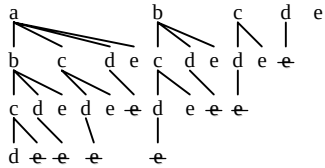

Figure 3: The tree for counting support used by Mafia

Items *c,d,e* are frequent in $T(ab)$. Item *d* is frequent and *e* infrequent in $T(abc)$. Mafia requires total 30 frequency testing. Using tail information to augment dynamic reordering, SmartMiner needs only 23 such frequency testing.

Finally, all previous approaches require superset checking for two purposes: pruning nodes and removing non-maximal itemsets in

MFI. If the set of MFI is large, as in most real dataset, the superset checking can be very expensive. In above example, Mafia performs 30 superset checking. As will be discuss later, SmartMiner does not require any superset checking.

2. Partition and Pruning Properties

In this section, we define some concepts for SmartMiner.

2.1 Partitioning a search space

Let $N=X:Y$ be a node where X is the head of N and Y is the tail of N. All possible subsets of Y is called the power set of Y, denoted by $P(Y)$.

Definition 1 For a node $N=X:Y$, the set of all the itemsets obtained by concatenating X with the itemsets in $P(Y)$ is called the **search space** of N, denoted as $\{X:Y\}$. That is

$$\{X:Y\} = \{X \cup V \mid V \in P(Y)\}.$$

For example, the search space $\{b:cd\}$ includes four itemsets b, bc, bd, and bcd. The search space $\{:abcde\}$ includes all subsets of $abcde$.

By definition 1, we have $\{X:Y\}=\{X:Z\}$ where $Z=Y-Z$. Thus we will assume Y does not contain any item in X when $\{X:Y\}$ is mentioned in this paper.

Definition 2 Let S, S_1, and S_2 be search spaces. The set $\{S_1, S_2\}$ is a *partition* of S if and only if $S= S_1 \cup S_2$ and $S_1 \cap S_2=\phi$. The relationship is denoted by $S=S_1+S_2$ or $S_1= S-S_2$ or $S_2= S-S_1$. We say S is partitioned into S_1 and S_2. Similarly, a set $\{S_1, S_2, \ldots, S_k\}$ is a partition of S if and only if $S= S_1 \cup S_2 \cup \ldots \cup S_k$ and $S_i \cap S_j=\phi$ for $i,j \in [1..k]$ and $i \neq j$. We denote it as $S=S_1+S_2+\ldots+S_k$.

Let a be an item, aX is an itemset by adding a to X.

Theorem 1 For $a \notin X,Y$, the search space $\{X:aY\}$ can be partitioned into $\{Xa:Y\}$ and $\{X:Y\}$ by item a, i.e., $\{X:aY\}=\{Xa:Y\}+\{X:Y\}$.

Proof: It follows from the fact that each itemset of $\{X:aY\}$ either contains a or does not.

For example, we have $\{b:cd\}=\{bc:d\}+\{b:d\}$.

In general, suppose $a_1,a_2,\ldots,a_k$ be distinct items and $a_1a_2\ldots a_k$Y be an itemset.

Theorem 2 *Partition search space*: the search space of $\{X: a_1a_2\ldots a_k\text{Y}\}$ can be partitioned into

$$\sum_{i=1}^{k}\{Xa_i : a_{i+1}\ldots a_k Y\}+\{X:Y\}, \text{ where } a_i \notin X,Y.$$

Proof: It follows by partitioning the search space via items $a_1,a_2,\ldots,a_k$ sequentially as in theorem 1.

For example, we have $\{b:cd\}=\{bc:d\}+\{bd:\}+\{b:\}$ and $\{a:bcde\}= \{ab:cde\} +\{ac:de\}+\{a:de\}$.

Let $\{X:Y\}$ be a search space and Z be a known frequent itemset. Since Z is frequent, all subset of Z will be frequent, i.e. every itemset of $\{:Z\}$ is frequent. Theorem 3 shows how to prune the space $\{X:Y\}$ by Z.

Theorem 3 *Pruning search space*: if Z does not contain the head X, the space $\{X:Y\}$ can not be pruned by Z, i.e., $\{X:Y\}$-$\{:Z\}=\{X:Y\}$. Otherwise, the space can be pruned as

$$\{X:Y\}\text{-}\{:Z\} = \sum_{i=1}^{k}\{Xa_i : a_{i+1}\ldots a_k (Y \cap Z)\},\ a_1a_2\ldots a_k=Y\text{-}Z.$$

Proof: If Z does not contain X, no itemset in $\{X:Y\}$ is subsumed by Z. Therefore, knowing Z frequent can not prune away any part of the search space $\{X:Y\}$. Otherwise X is a subset of Z, we have

$$\{X:Y\}= \sum_{i=1}^{k}\{Xa_i : a_{i+1}\ldots a_k V\} + X:V, \quad \text{where} \quad V=\text{Y} \cap \text{Z}.$$

The head in the first part is Xa_i. Since Z does not contain a_i, the first part can not be pruned. For the second part, we have $\{X:V\}$-$\{:Z\}=\{X:V\}$-$\{X:(Z\text{-}X)\}$. Since $\text{X} \cap \text{Y}=\phi$, we have $V \subseteq Z\text{-}X$. Therefore $\{X:V\}$ can be pruned away entirely.

For example, we have $\{:bcde\}$-$\{:abcd\}=\{:bcde\}$-$\{:bcd\}=\{e:bcd\}$. And $\{e:bcd\}$-$\{:abe\}=\{e:bcd\}$-$\{:be\}= \{e:bcd\}$-$\{e:b\} = \{ec:bd\}+\{ed:b\}$.

2.2 Evaluating Tail Information

Definition 3 Let M be known frequent itemsets and $N=X:Y$ be a node. The **tail information** of M to N, $TInf\,(N|M)$, is the tail parts of the frequent itemsets in $\{X:Y\}$ that can be inferred from M,i.e.,

$$TInf\,(N \mid M) = \{Y \cap Z \mid \forall Z \in M, X \subseteq Z\}$$

For example, $TInf\ (e:bcd|\{abcd,abe,ace\})=\{b,c\}$, which means that eb and ec are frequent given $\{abcd,abe,ace\}$ frequent. $Inf(e:bcd|\{abcd,abe,ace,bce\})=\{b,c,bc\}$. For simplicity we call tail information as information.

Definition 4 The value of tail information W is all itemsets that are subsets of some member of W. That is,

$$VTI(W) = \{X \mid \forall Z \in W, X \subseteq Z\}$$

For example, $VTI(\{b,c,bc\})=\{\phi,b,c,bc\}=VTI(\{bc\})$. Notice that removing non-maximal itemsets from information set does not decrease its value. Therefore whenever we found a non-maximal itemset in the information set, we deleted it.

3. The Strategy of SmartMiner

3.1 Tail information guided depth-first search

Assume the tail of a node may contain many infrequent items, pure depth-first search is inefficient. Hence dynamic reordering is used to prune away infrequent items from the tail of a node before exploring its sub nodes.

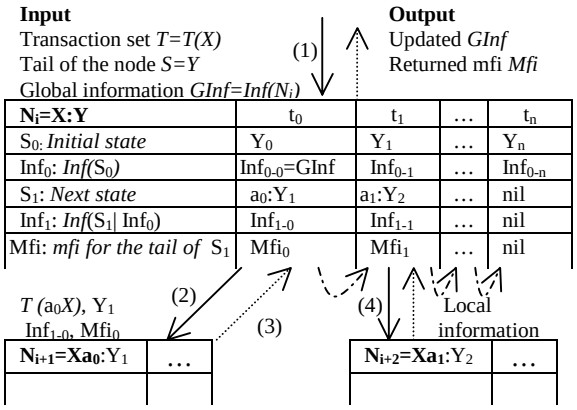

Figure 4: Search strategy illustrated at the node N_i=*X:Y*

SmartMiner uses tail information to guide depth-first search. We illustrate the strategy for a given node N_i=*X:Y* as shown in Figure 4. The purpose of the node N_i=*X:Y* is to compute maximal frequent itemsets in the transaction set *T(X)*. The inputs for node N_i=*X:Y* are transaction set *T(X)*, the tail *Y*, and the tail information for N_i known so far, *Ginf*, is called global tail information for node N_i. The outputs of the node are the updated *GInf* and discovered maximal frequent itemsets *Mfi*. Upon calling the node N_i, we count the supports for the items in the tail *Y*. By removing infrequent items from *Y*, we have Y_0.

The time sequence at node N_i in Figure 4 is $t_0,t_1,\ldots,t_n$. At the moment t_0, item a_0 is selected from Y_0 to be the head of next state S_1 and Y_1= Y_0- a_0 is the tail of S_1. The tail information $Inf_{1\text{-}0}$ is computed by *Inf*(a_0:Y_1 |*GInf*). We then create node N_{i+1}=Xa_1:Y_1. The call for node N_{i+1} returns Mfi_0 and updated $Inf_{1\text{-}0}$ in which the members subsumed by Mfi_0 are marked deleted. At t_1, we calculate the tail information $Inf_{0\text{-}1}$ for Y_1 from $Inf_{0\text{-}0}$, $Inf_{1\text{-}0}$, and Mfi_0. The information from $Inf_{0\text{-}0}$ and $Inf_{1\text{-}0}$ is updated global information. The information from Mfi_0 is local information. Using information $Inf_{0\text{-}1}$, item a_1 is selected from Y_1 to be the head of the next state S_1 and Y_2= Y_1- a_1 is the tail of S_1. Then node N_{i+2}=Xa_2:Y_2 is created and called to compute maximal frequent itemsets in transaction sets $t(Xa_2)$. This process continues till t_n where no item can be selected as head of S_1. The returned maximal frequent itemsets $Mfi = \cup a_i Mfi_i, i \in [0..n\text{-}1]$; the updated *GInf* is these itemsets in the original *GInf* which have not marked as deleted.

SmartMiner uses tail information to guide depth-first search which is different from dynamic reordering depth-first strategies (DFS). First, SmartMiner defers creating a node till its preceding nodes are visited, while DFS creates nodes for each item in the tail of a node in the increasing order of their supports. DFS creates as many sub trees as the number of frequent items in the tail. Second, SmartMiner augments the dynamic ordering heuristic with considering the tail information about each item (see section 4.3). Using this heuristic, SmartMiner creates far less sub trees than simple dynamic reordering. Finally, by passing tail information, SmartMiner does not require the time for superset checking that is required for DFS.

3.2 An example

We now use an example to illustrate how *SmartMiner* finds the same *MFI* as shown in Figure 5 for the problem in Figure 2. There are nine nodes N_0, N_1, … , N_8 in the search tree. For a given node, the columns t_0, t_1, …, t_m represent the sequential time point of the node. The row S_0 represents the initial state and the Inf_0 is the tail information for S_0. The row S_1 is the next state to explore and the relevant information is on the row Inf_1. Note here Inf_1 also called the global information as input for the next state and will be updated. The row *Mfi* is the returned *mfi* after exploring the state S_1. On top of each node, we give the transaction set for the node. For example, the transaction set for N_0 is the entire dataset $T(\phi)$; the transaction set for N_1 is $T(a)$ which represents all the transactions containing item *a*.

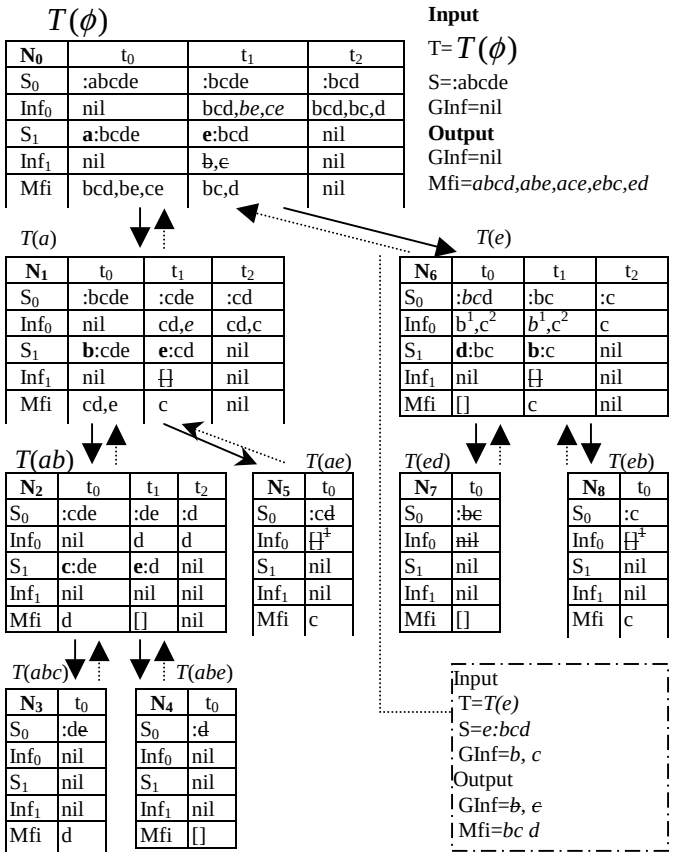

Figure 5: An example of using SmartMiner to discover the MFI

SmartMiner begins at the node N_0 at t_0, $N_0(t_0)$, where S_0=*:abcde* and Inf_0 is empty. At this point, item *a* is selected and thus the next state S_1=*a:bcde*. Here Inf_1 is empty since Inf_0 is empty. Next *SmartMiner* create the node N_1 for the state S_1=*a:bcde* by setting its transaction set $T(a)$ and its initial set S_0=*:bcde*. When *SmartMiner* call the new node N_1, each item in the tail S_0=*:bcde* will be sorted in the increasing order of their support in $T(a)$ and the infrequent items will be dropped. The process continues to $N_2(t_0)$, and then to $N_3(t_0)$ where S_0=*:de* and *e* is dropped since it is infrequent in $T(abc)$. This yields S_0=*:d*, *SmartMiner* returns *d* as *mfi* to $N_2(t_0)$ which will be added into

Inf_0 at $N_2(t_1)$. Thus at $N_2(t_0)$, $Inf_0 = d$. *SmartMiner* then select S_1=*e:d* for next node, $N_4(t_0)$.

The entire search route will be $N_0(t_0)$, $N_1(t_0)$, $N_2(t_0)$, $N_3(t_0)$, $N_2(t_1)$, $N_4(t_0)$, $N_2(t_2)$, $N_1(t_1)$, $N_5(t_0)$, $N_1(t_2)$, $N_0(t_1)$, $N_6(t_0)$, $N_7(t_0)$, $N_6(t_1)$, $N_8(t_0)$, $N_6(t_2)$, and $N_0(t_2)$. As shown in the figure, at $N_0(t_1)$, Inf_0=*bcd,be,ce*, S_1=*e:bcd*, and the two itemsets *be,ce* contain *e*. By removing *e* from *be,ce*, we get Inf_1=*b,c*. When calling N_6, global information *Ginf=b,c* is passed from $N_0(t_1)$ to $N_6(t_0)$. Upon completing exploring the node N_6, *bc,d* are found to be *mfi* and *Ginf=b,c* will be updated to be empty since they are dropped respectively at $N_8(t_0)$ to $N_6(t_1)$ and at $N_6(t_1)$ to $N_6(t_2)$. When it returns from N_6, the Inf_1 at $N_0(t_1)$ will be empty. By collecting *Mfi*, Inf_1, and unselected Inf_0 at $N_0(t_1)$, we have Inf_0=*bcd,bc,d* at $N_0(t_2)$. The search terminates at $N_0(t_2)$ since the tail of S_0=*:bcd* is in the Inf_0.

Figure 6 shows the tree for counting support using *SmartMiner*. At node N_0, *SmartMiner* counts the supports for *a,b,c,d,e* and found they are frequent. At node N_1, items *b,c,d,e* are found to be frequent in *T(a)*. It is shown that there are a total of 23 times to count for support.

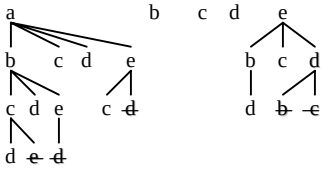

Figure 6: The tree for counting support used by SmartMiner

4. Algorithmic Descriptions

4.1 Object model design

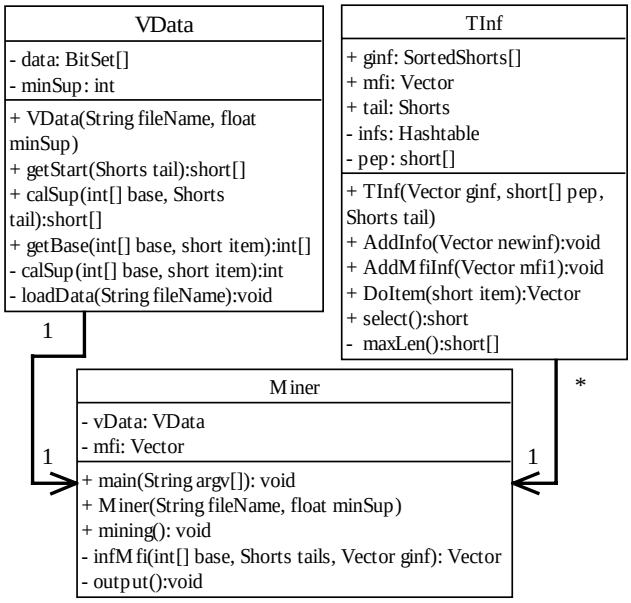

Figure 7: The object model used for implementing SmartMiner

Our data mining system is implemented in Java rather than C++ because Java has better portability. Figure 7 shows the three classes in our system whose data types are specified using Java language. The class *VData* is the vertical data model for a transaction dataset. It loads data from a given *fileName* and builds up a *BitSet* for each frequent item. The *TInf* class manages the tail information for a given node. The *Miner* class uses the proposed tail information based depth-first search to recursively discover all *MFI*. An instance of *Miner* has exactly one object of *VData* and will dynamically create one object of *TInf* for a node when the mining starts. More details is given in the following sections.

4.2 Vertical data class: VData

We chose to use a vertical *BitSet* representation for the database. A vertical *BitSet* corresponds to one frequent item. In a *BitSet*, there is one bit for each transaction in the database. If item *i* appears in transaction *j*, then bit *j* of the *BitSet data*[*i*] is set to one; otherwise, the bit is set to zero. The constructor *VData(String filename, float minSup)* calls the private function *Load(String filename)* to load data from the file into the variable *data*. It also calculates the *minSup* by multiplying the float *minSup* with the number of transactions. The variable *int[] base* in methods *calSup* and *getBase* is an array of transaction id. The base of a node represents the transaction set *T(X)* where *X* is the head of the node. The private method *calSup(int[] base, short item):int* is to calculate the support of the *item* in the given *base*.

The *VData* provides three methods for data mining. First, the method *getStart(Shorts tail):short[]* returns the set of items that occur in every transaction. It also passed other items by *Shorts tail* in the order of increasing support. The *getStart* is called at a root node. Second, the public method *calSup(int[] base, Shorts tail):short[]* is similar to the *getStart*. It returns the set of items in every transaction of the *base* and passes other frequent items at the *base* in the order of increasing support. Finally, the method *getBase(int[] base, short item):int[]* simply returns a new base which is the subset of the *base* whose corresponding transactions contains the *item*.

Note that when calculating support of an item in a *base*, the *VData* needs to test as many bits as the size of the *base*. It is slower than the *Bitmap* model where supports can be calculated a byte (8 bits) at a time. Our *VData* model is also slower than the *diffset* model of *GenMax*[] . However, the *VData* keeps only one copy of data and thus needs less memory than the other two models. In other words, both *Mafia* and *GenMax* need to build up new datasets for the mining of sub nodes. Moreover, the *VData* is easy to implement and is fair to use it as a common data model to compare different search strategies of *SmartMiner*, *Mafia*, and *GenMax*.

4.3 Tail information class: TInf

For a given node, an instance of the *TInf* class is created to manage the tail information at the node. The global information *ginf* is passed from its parent node. The *mfi* is the local maximal frequent itemsets discovered at the node. The itemsets to be explored is stored in the *tail*. Tail information for the *tail* is stored in the hash table *infs*. The *pep* is the items occurred in every transaction of the transaction set of the node which is specified by the *base*.

The constructor method accepts global information *ginf*, common items *pep*, and a *tail* to create a new instance. The public methods *AddInfo* and *AddMfiInf* calculate relevant information of the *newinf* and the *mfi1* on *tail* respectively and then hash them into the hash table *infs*. The method *DoItem(short item):Vector* separates the itemsets in the *infs* into two groups: one mentions the *item*; another does not. The first group will be removed from

the hash table and returned as a vector after dropping the *item* from its itemsets. The second group remains in the table. The method also removes the *item* from the *tail*. For every item in the *tail*, the private method *maxLen():short[]* is to find the maximal length of itemsets in the hash table *infs* that contains the *item*. Note that, in our experiment, we use a simplified *maxLen* that returns an array of value either 0 or the maximal length. More specifically, the *maxLen* first finds the longest itemset *V* in the *infs* and then set the lengths of items in *V* to |*V*| and the lengths of other items to 0.

Figure 8 describes the selection method which is a heuristic to select an item to partition the search space. In dynamic reordering, the item of least support is chosen to explore first since it is likely that the sub search tree is small. This heuristic is shown to be very effective. We augument it by the observation that, if an item contained by an itemset of size *k* in the *infs*, there are 2^k itemsets that are known to be frequent and can be pruned away from the search space. Therefore our heuristic chooses an item of the smallest known space, i.e., not occur in long itemsets in the *infs*. If the size of current tail is less than 2, the search space is immediately solvable as shown in line 1~3. Line 4 calls the method *maxLen*. Line 5 is to find the positions of the minimal and maximal values in *len*. Note that, if there are several positions for minimal value, we will choose the least position of them since the corresponding item in the *tail* has the least support. If there is an itemset in the *infs* has the size of the *tail*, this means the whole search space of the *tail* is frequent and thus there is no need to build a sub node as shown in line 6-8. If there are some itemsets originated from *ginf* and they are not of the size of the *tail*, the corresponding itemsets in *ginf* will be deleted since they are subsets of some other itemset. Line 9 returns the selected item.

```
/**
 * Select an item to build a sub node.
 * @return  >=0 if success, -1 if no next items.
 */
public short select()
1 if(tail.size()<=1)
2   if tail in infs then mfi=null else mfi=tail;
3   return -1;
4 short[] len = maxLen();
5 find the min, max position minp, maxp in len;
6 if(len[maxp]==tail.size())
7   update the ginf info;
8   return -1;
9 return tail.get(minp);
```

Figure 8: The selection method: a heuristic to select an item for partitioning the search space.

4.4 Data mining class: Miner

The *Miner* class has two attributes and five methods as shown in Figure 7. The *vData* is an instance of the class *VData*. It stores transaction data in vertical format. The *mfi* is a vector of maximal frequent itemsets. The *main* method reads *filename* and *minSup* from command line and calls methods *Miner*, *mining*, and *output* sequentially. The *Miner* builds an instance of this class and initializes *vData*. The output method simply writes the *mfi* into a file. The *mining* method is to mine the *vData*.

Now we present the information guided depth first algorithm as in Figure 9. The parameter *base* is the transaction set for the head of current node. The *tail* is the possible extension of the head. The *ginf* is the globe information passed to the node. Note that *ginf* is a reference parameter, whose value can be updated. The method returns local maximal frequent itemsets. Line 1 calls *vData.calSup* to get the *pep* and an updated *tails* sorted in the increasing order of support. Line 2 creates an instance of the *Information* class for this node. Lines 3~8 loop selects an item for next node and make calls recursive call. More specifically, it selects an item *itm* for next node as show in line 3. If there is no node selected, it goes to line 9. Otherwise, it enters the loop body. A new base is calculated at line 4; the *inf.DoItem* method is called; and the new_tail is set. Then line 7, calls the selected sub node. Upon returning from the sub node, it adds the updated *new_ginf* into the *inf* at line 8 and also saves the *new_mfi* by method *AddMfiInf* at line 9. It returns the *mfi* of the node at line 10.

```
/**
 * Recursively find mfi.
 * @param base The tidSet for current head.
 * @param tail The possible extension of the head.
 * @param ginf The global information.
 * @return  The local maximal frequent itemsets.
 */
private Vector infMfi(int[] base, Shorts tails,
                      Vector ginf)
1 short[] pep = vData.calSup(base,tails);
2 TInf inf = new TInf(ginf, pep, tails);
3 while((itm=inf.select())>=0)
4   int[] newbase = vData.getBase(base,itm);
5   Vector newginf=inf.DoItem(itm);
6   Shorts newtail=new Shorts(inf.tail);
7   Vector newmfi=infMfi(newbase,newtail,newginf);
8   inf.AddInfo(newginf);
9   inf.AddMfiInf(newmfi);
10 return inf.mfi;
```

Figure 9: The infMfi method--the tail information guided depth-first search

For the node at the level 0, the local *new_mfi* is actually maximal frequent itemsets and can output directly into a file. Since its information for future searching is saved by the method *inf.AddMfiInf* in line 9, there is no need to keep the *new_mfi* and the memory of *new_mfi* can be released.

5. Experimental Results

We compare SmartMiner with Mafia and GenMax. All of them are implemented in Java JDK1.3. For fair comparison, the three methods use the same vertical data model VData. As we discussed before, there are many ways to implement vertical data model. In this paper, our purpose is to study the efficiency of different search strategies and we are not interested in comparing the different data models. We choose VData since it takes less memory and it is easy to implement. The experiment was performed on a 1Ghz Celeron with 512 MB of memory running Microsoft Windows 2000 Professional. SmartMiner was tested with two datasets: connect-4 and mushroom. A detailed comparison of SmartMiner on these datasets with Mafia and GenMax was conducted.

Figure 10 shows the performance comparison of the three methods on Mushroom. All the three methods implement the PEP pruning technique. Our running time does not include the input time but does include the output time. The horizontal axis shows minimum support in percentage. The vertical axis is the

running time in seconds. In general, SmartMiner is one order of magnitude faster than both Mafia and Genmax. When minimal support is high, Mafia is faster than Genmax. Low minimal support increase the number of MFI, then Genmax performs better than Mafia.

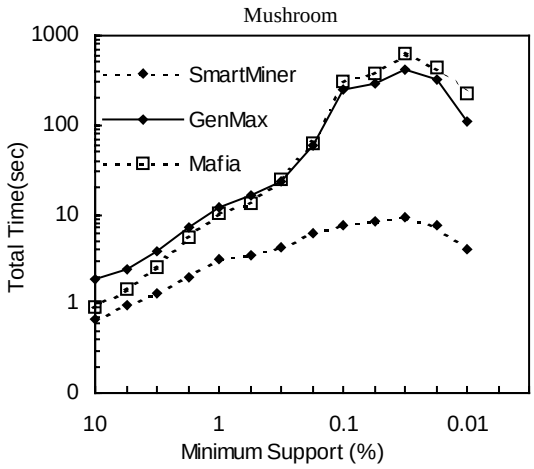

Figure 10: Performance comparison on Mushroom for selective minimum supports.

Figure 11 compares the sizes (number of nodes in a tree) of the search trees for the three methods. From the figure, we notice that Genmax generates 10 times more nodes than SmartMiner and also much more than Mafia. This indicates that the static ordering in GenMax is not as efficient as the dynamic reordering used by both SmartMiner and Mafia. Moreover, we notice that SmartMiner generates less nodes than Mafia, which reveals that our augmented heuristic is better than a pure dynamic reordering.

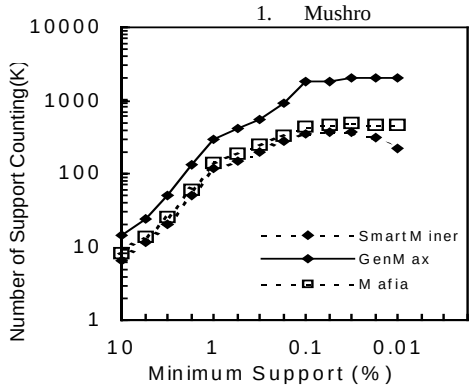

Figure 12: Comparison of the # of support counting for selective minimum support.

Figure 12 compares the number of support counting which shows the number of times that the private method calSup(int[] base, short item) in VData is called. As in Figure 12, Genmax calls the calSup methods significantly more than both SmartMiner and Mafia. Further, SmartMiner needs less number of support counting than Mafia.

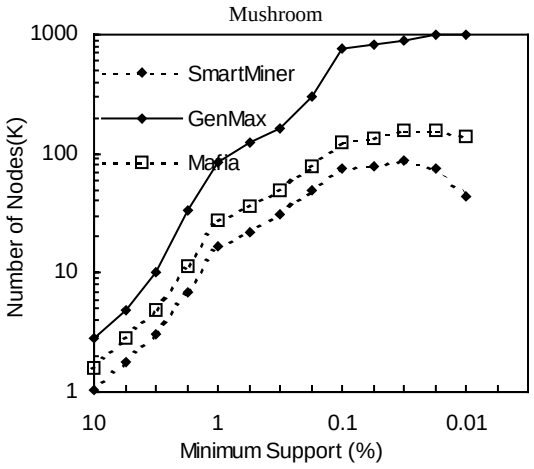

Figure 11: Comparison of tree size on mushroom for selective minimum support.

Since GenMax introduces a fast superset checking algorithm, the performance gain of dynamic reordering of Mafia is mitigated by the increasing time for superset checking when the set of MFI becomes large. This is the reason we see in Figure 10 and Figure 13 that Mafia is better than Genmax when minimal support is high and the reverse when minimal support is low.

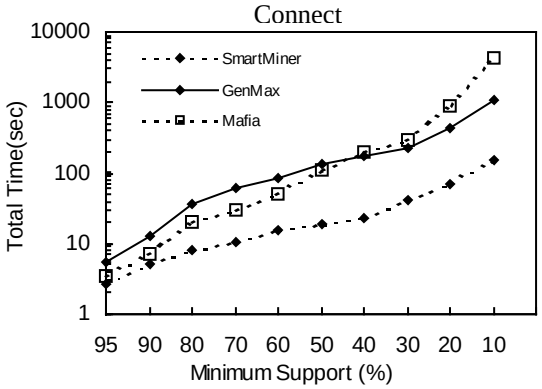

Figure 13: Performance comparison on mushroom for selective minimum support.

Figure 13 shows the performance comparison of the three methods for Connect dataset. Again, we notice the significant performance improvements of SmartMiner.

6. Conclusion

In this paper, we propose the SmartMiner algorithm to find exact maximal frequent itemsets for large datasets. The SmartMiner algorithm first uses global and local tail information to augment dynamic reordering to reduce the search tree. Second, the passing of tail information eliminates the need of known MFI for superset checking. Smartminer does require superset checking that can be very expensive. Finally, SmartMiner also reduces the number of support counting for determining the frequency of tail items and thus greatly saves counting time. Our experiments reveal that the SmartMiner algorithm yields an order of magnitude improvement over the Mafia and GenMax in generating the MFI for the same datasets.

ACKNOWLEDGMENTS

The authors wish to thank Professor Mohammed J. Zaki at Rensselaer Polytechnic Institute for his help in the performance study.

REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In Proceedings of the 20th VLDB Conference, Santiago, Chile, 1994.

[2] R. Agarwal, C. Aggarwal and V. Prasad. A tree projection algorithm for generation of frequent itemsets. Journal of Parallel and Distributed Computing, 2001.

[3] Roberto Bayardo. Efficiently mining long patterns from databases. In ACM SIGMOD Conference, 1998.

[4] D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: a maximal frequent itemset algorithm for transactional databases. In Intl. Conf. on Data Engineering, Apr. 2001.

[5] K. Gouda and M. J. Zaki. Efficiently Mining Maximal Frequent Itemsets. Proc. of the IEEE Int. Conference on Data Mining, San Jose, 2001.

[6] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation, Proc. 2000 ACM-SIGMOD Int. Conf. on Management of Data (SIGMOD'00), Dallas, TX, May 2000.

[7] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Efficient algorithms for discovering association rules. In KDD-94: AAAI Workshop on Knowledge Discovery in Databases, pages 181-192, Seattle, Washington, July 1994

[8] J. S. Park, M. Chen, and P. S. Yu. An effective hash based algorithm for mining association rules. In Proc. ACM SIGMOD Intl. Conf. Management of Data, May 1995.

[9] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In 7th Intl. Conf. on Database Theory, January 1999.

[10] J. Pei, J. Han, and R. Mao. Closet: An efficient algorithm for mining frequent closed itemsets. In SIGMOD Int'l Workshop on Data Mining and Knowledge Discovery, May 2000.

[11] Brin, S.; Motwani, R.; Ullman, J.; and Tsur, S. 1997. Dynamic Itemset Counting and Implication Rules for Market Basket Data. In Proc. of the 1997 ACM-SIGMOD Conf. On Management of Data, 255-264.

[12] Ashok Sarasere, Edward Omiecinsky, and Shamkant Navathe. An efficient algorithm for mining association rules in large databases. In 21st Int'l Conf. on Very Large Databases (VLDB), ZTrich, Switzerland, Sept. 1995.

[13] Hannu Toivonen. Sampling large databases for association rules. In Proc. of the VLDB Conference, Bombay, India, September 1996.

[14] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In 3rd Intl. Conf. on Knowledge Discovery and Data Mining., August 1997.

[15] M. J. Zaki and C. Hsiao. Charm: An efficient algorithm for closed association rule mining. In Technical Report 99-10, Computer Science, Rensselaer Polytechnic Institute, 1999.

[16] Q. Zou, W. Chu, D. Johnson and H. Chiu. A Pattern Decomposition (PD) Algorithm for Finding All Frequent Patterns in Large Datasets. Proc. of the IEEE Int. Conference on Data Mining, San Jose, 2001.

[17] Q. Zou, W. Chu, D. Johnson and H. Chiu. Pattern Decomposition Algorithm for Data Mining of Frequent Patterns. Journal of Knowledge and Information System {to appear}, 2002.